Greedy Algorithms

Topics to be Covered

- General Characteristics of greedy algorithms
- Elements of Greedy Strategy
- Make change Problem
- Minimum Spanning trees (Kruskal’s algorithm, Prim’s algorithm)
- The Knapsack Problem
- Job Scheduling Problem
- Huffman code.

Characteristics of Greedy Algorithms

- Greedy algorithms are characterized by the following features.
    1. Greedy approach forms **a set or list of candidates C**.
    2. Once a candidate is **selected in the solution**, it is there forever: once a candidate is **excluded from the solution**, it is never reconsidered.
    3. To construct the solution in an optimal way, Greedy Algorithm maintains **two sets**.
    4. One set contains candidates that have already been **considered and chosen**, while the other set contains candidates that have been **considered but rejected**.

Elements of Greedy Strategy

- The greedy algorithm consists of **four functions**.
  1. **Solution Function**:- A function that checks whether chosen set of items provides a solution.
  2. **Feasible Function**:- A function that checks the feasibility of a set.
  3. **Selection Function**:- The selection function tells which of the candidates is the most promising.
  4. **Objective Function**:- An objective function, which does not appear explicitly, but gives the value of a solution.

Make Change Problem

- Suppose following coins are available **with unlimited quantity**:
  1. ₹ 10
  2. ₹ 5
  3. ₹ 2
  4. ₹ 1
  5. 50 paisa

- Our problem is to devise an algorithm for paying a given amount to a customer using **the smallest possible number of coins**.

Make Change Problem

- If suppose, we need to pay an amount of ₹ 28/- using the available coins.
- Here we have a candidate (coins) set $\boldsymbol{C} = \{10, 5, 2, 1, .5\}$
- The greedy solution is,

Selected coins are,

coins	quantity
10	2
5	1
2	1
1	1

Amount

28

Total required coins = 5

Selected coins = {10, 5, 2, 1}

Make Change - Algorithm

```
# Input: C = {10, 5, 2, 1, 0.5} //C is a candidate set
# Output: S: set of selected coins
Function make-change(n): set of coins
S ← Ø {S is a set that will hold the solution}
sum ← 0 {sum of the items in solution set S}
while sum ≠ n do
    x ← the largest item in C such that sum + x ≤ n
    if there is no such item then
        return "no solution found"
    S ← S U {a coin of value x)
    sum ← sum + x
return S
```

Make Change – The Greedy Property

- The algorithm is **greedy** because,
  - At every step it chooses **the largest available coin**, without worrying whether this will prove to be a **correct** decision later.
  - It **never changes** the decision, i.e., once a coin has been included in the solution, it is there **forever**.
- Examples: Some coin denominations 50, 20, 10, 5, 1 are available.
  1. How many minimum coins required to make change for 37 cents?

     5
  2. How many minimum coins required to make change for 91 cents?

     4
  3. Denominations: $d_1=6$, $d_2=4$, $d_3=1$. Make a change of ₹ 8. ~~3~~

     The minimum coins required are 2

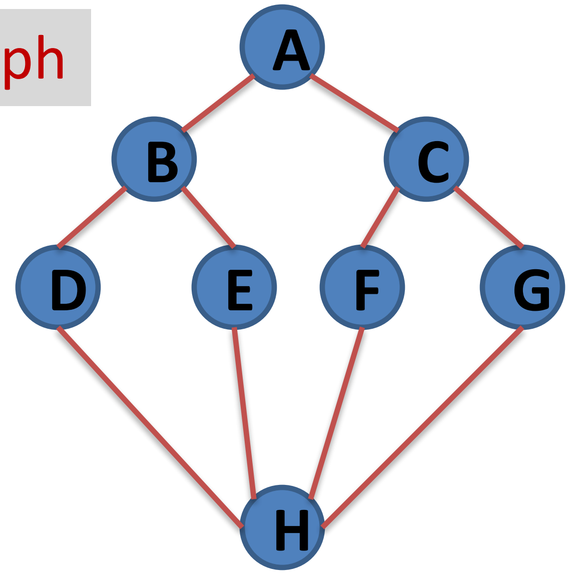
Minimum Spanning Tree

Minimum Spanning Tree (MST)

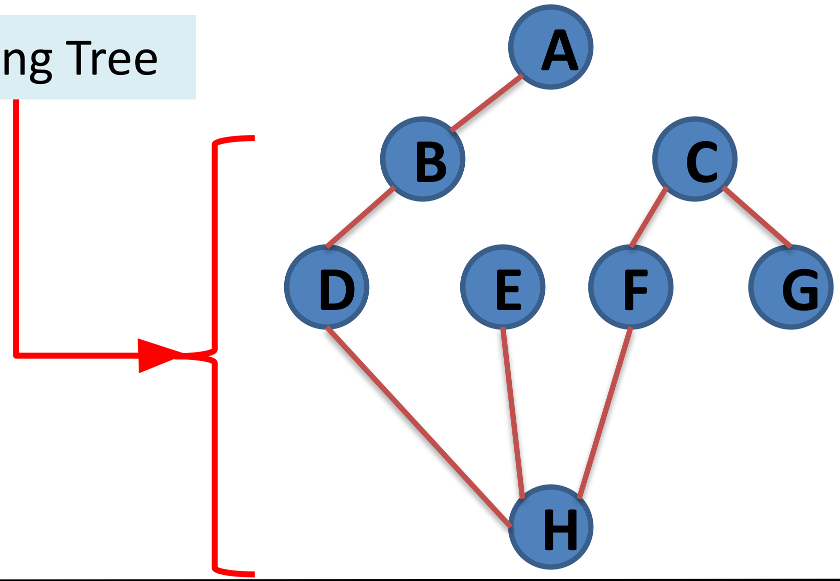
- Let $G = \langle N, A \rangle$ be a **connected, undirected graph** where,
  1. N is the set of nodes and
  2. A is the set of edges.
- Each edge has a given **positive length or weight**.
- A spanning tree of a graph G **is a sub-graph** which is basically a tree and it contains all the vertices of G but does not contain cycle.
- A minimum spanning tree (MST) of a **weighted connected graph** G is a spanning tree with minimum or smallest weight of edges.
- Two Algorithms for **constructing** minimum spanning tree are,
  1. Kruskal's Algorithm
  2. Prim's Algorithm

Spanning Tree

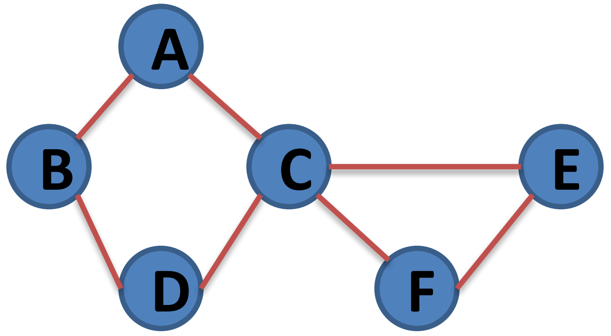
Graph



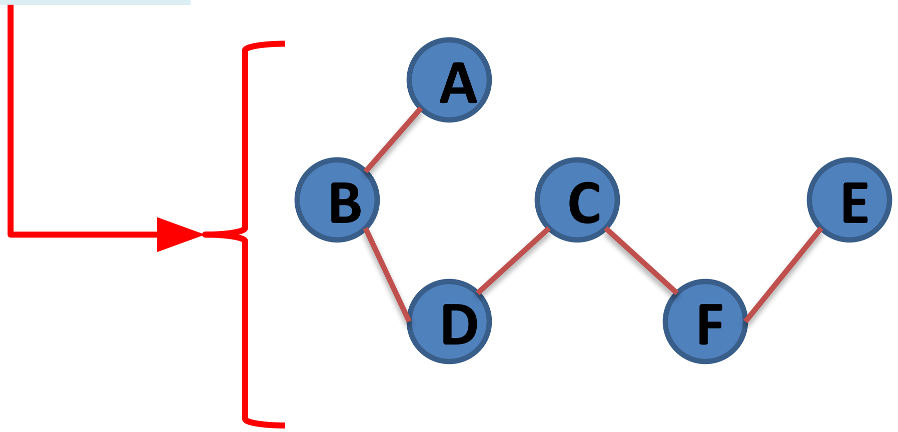
Spanning Tree



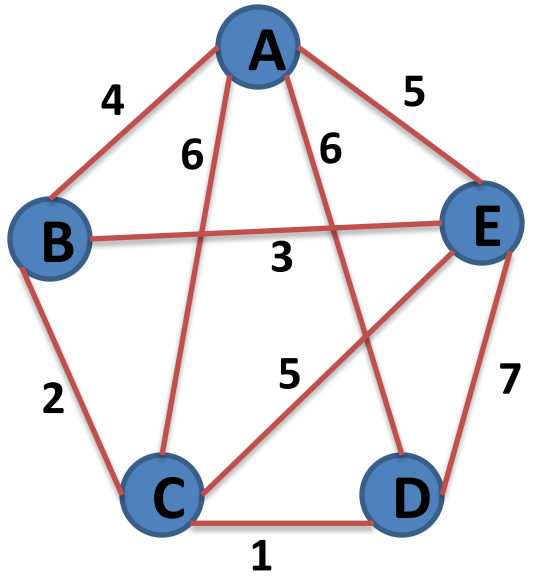
Graph



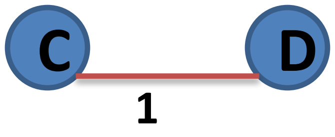
Spanning Tree



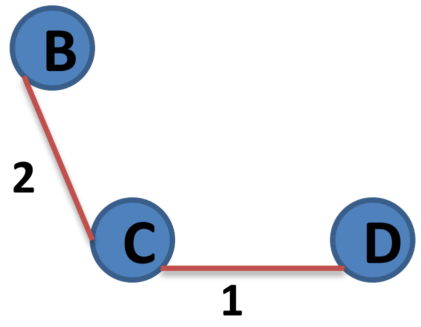
MST - Kruskal's Algorithm - Example



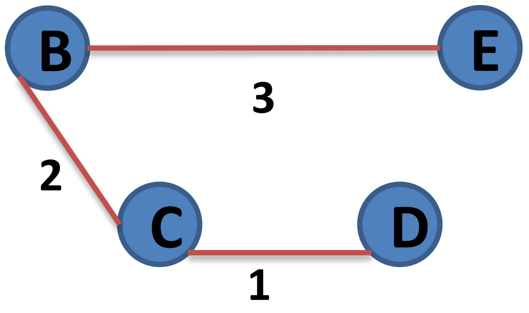
Step 1: Taking min edge (C,D)



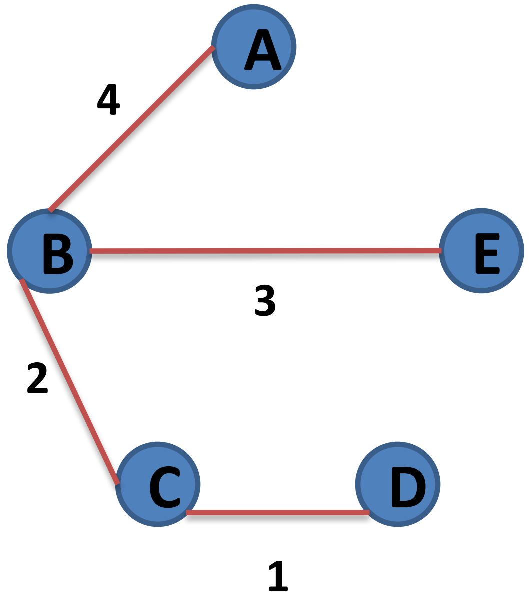
Step 2: Taking next min edge (B,C)



Step 3: Taking next min edge (B,E)



Step 4: Taking next min edge (A,B)



So, we obtained a minimum spanning tree of cost:

4 + 2 + 1 + 3 = 10

Kruskal's Algorithm

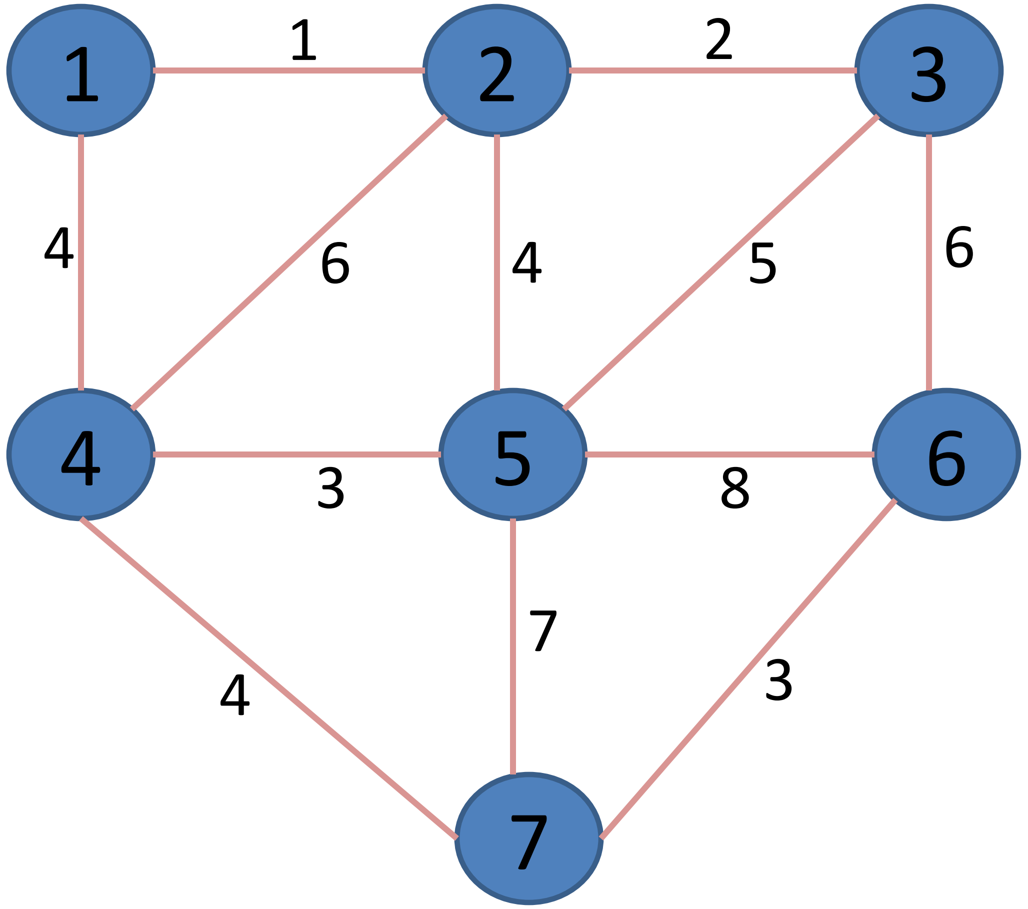
```
Function Kruskal(G = (N, A))
Sort A by increasing length
n ← the number of nodes in N
T ← ∅ {edges of the minimum spanning tree}
Define n sets, containing a different element of set N
repeat
    e ← {u, v} //e is the shortest edge not yet considered
    ucomp ← find(u)
    vcomp ← find(v)
    if ucomp ≠ vcomp then merge(ucomp, vcomp)
    T ← T U {e}
until T contains n - 1 edges
return T
```

find(u) tells in which connected component a node u is found

merge(ucomp, vcomp) is used to merge two connected components.

Kruskal's Algorithm - Example

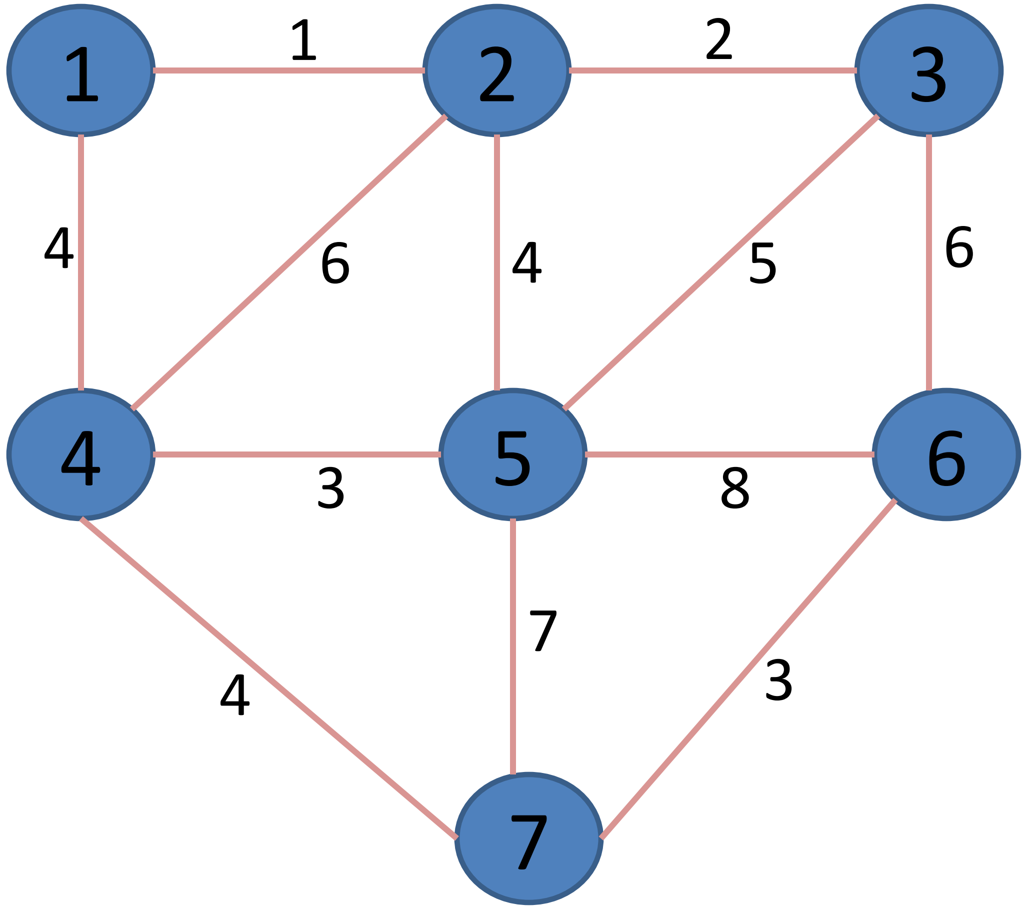
- Find the minimum spanning tree for the following graph using Kruskal's Algorithm.



Kruskal’s Algorithm - Example

Step:1

Sort the edges in increasing order of their weight.

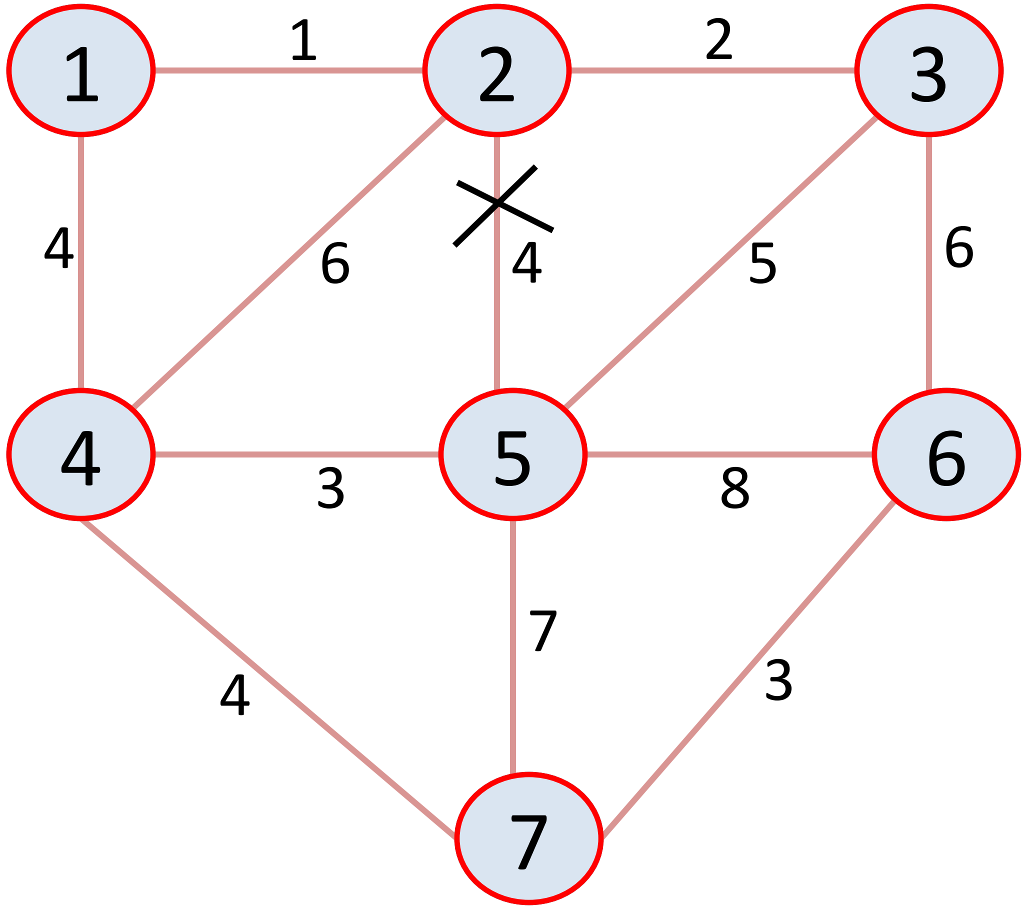


Edges	Weight	

Kruskal’s Algorithm - Example

Step:2

Select the minimum weight edge but no cycle.



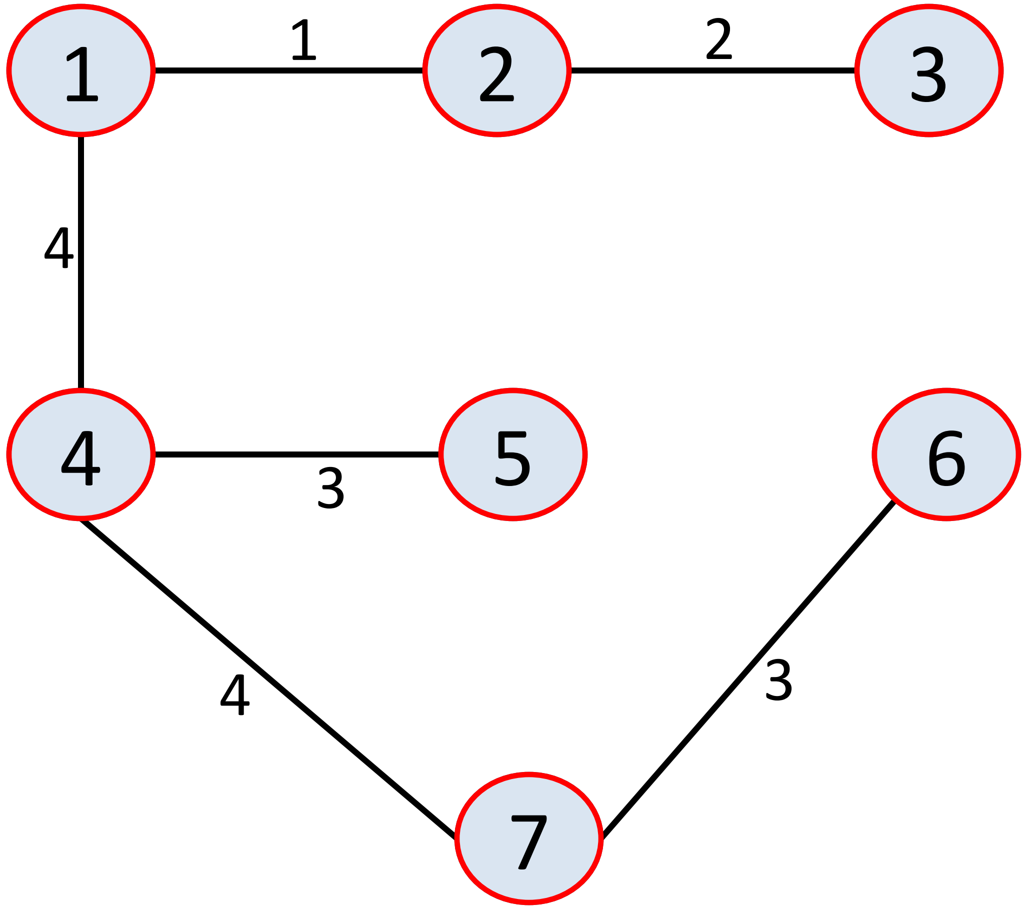
Edges	Weight	
{1, 2}	1	√
{2, 3}	2	√
{4, 5}	3	√
{6, 7}	3	√
{1, 4}	4	√
~~{2, 5}~~	~~4~~	
{4, 7}	4	√
{3, 5)	5	
{2, 4}	6	
{3, 6}	6	
{5, 7}	7	
{5, 6}	8	

Kruskal’s Algorithm - Example

Step:3

The minimum spanning tree for the given graph

Total Cost = 17



Kruskal’s Algorithm - Example

Step	Edges considered - {u, v}	Connected Components

Edges	Weight
{1, 2}	1
{2, 3}	2
{4, 5}	3
{6, 7}	3
{1, 4}	4
~~{2, 5}~~	~~4~~
{4, 7}	4

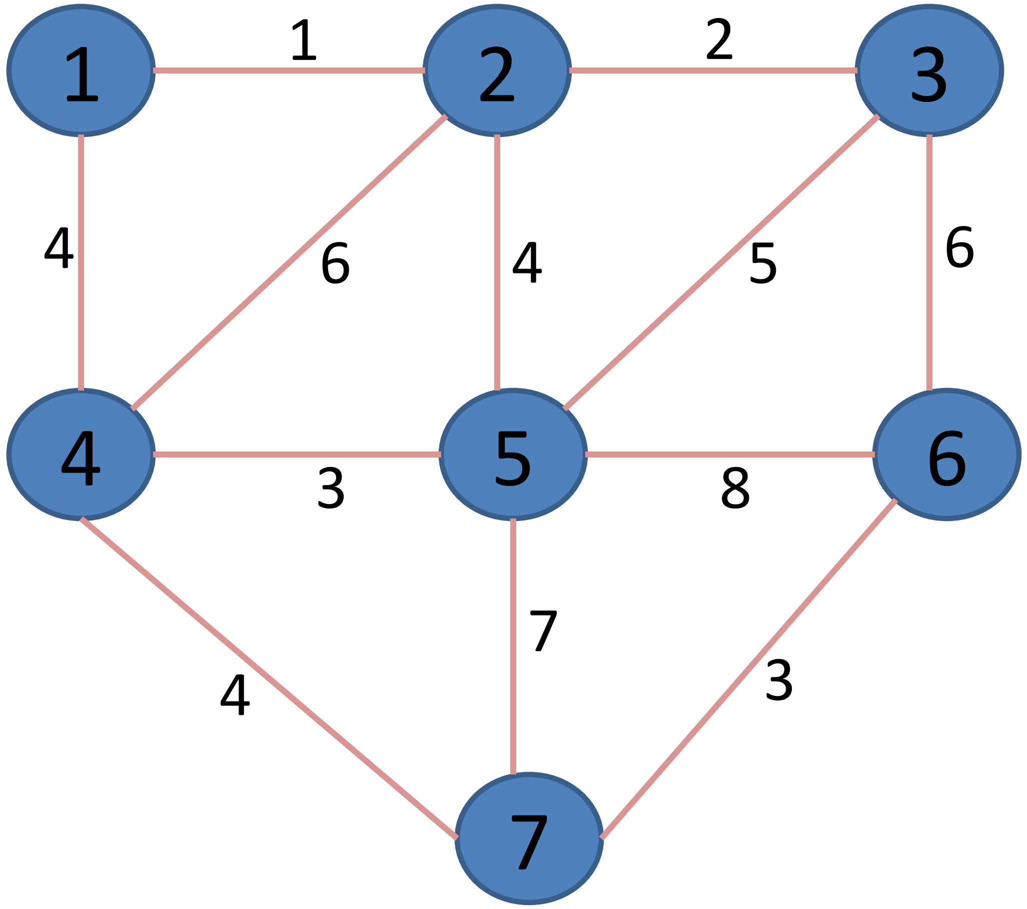
Total Cost = 17

Prim’s Algorithm

- In Prim's algorithm, the minimum spanning tree grows in a natural way, starting from an arbitrary root.
- At each stage we add a new branch to the tree already constructed; the algorithm stops when all the nodes have been reached.
- The complexity for the Prim’s algorithm is $\boldsymbol{\theta(n^2)}$ where n is the total number of nodes in the graph G.

Prim's Algorithm - Example

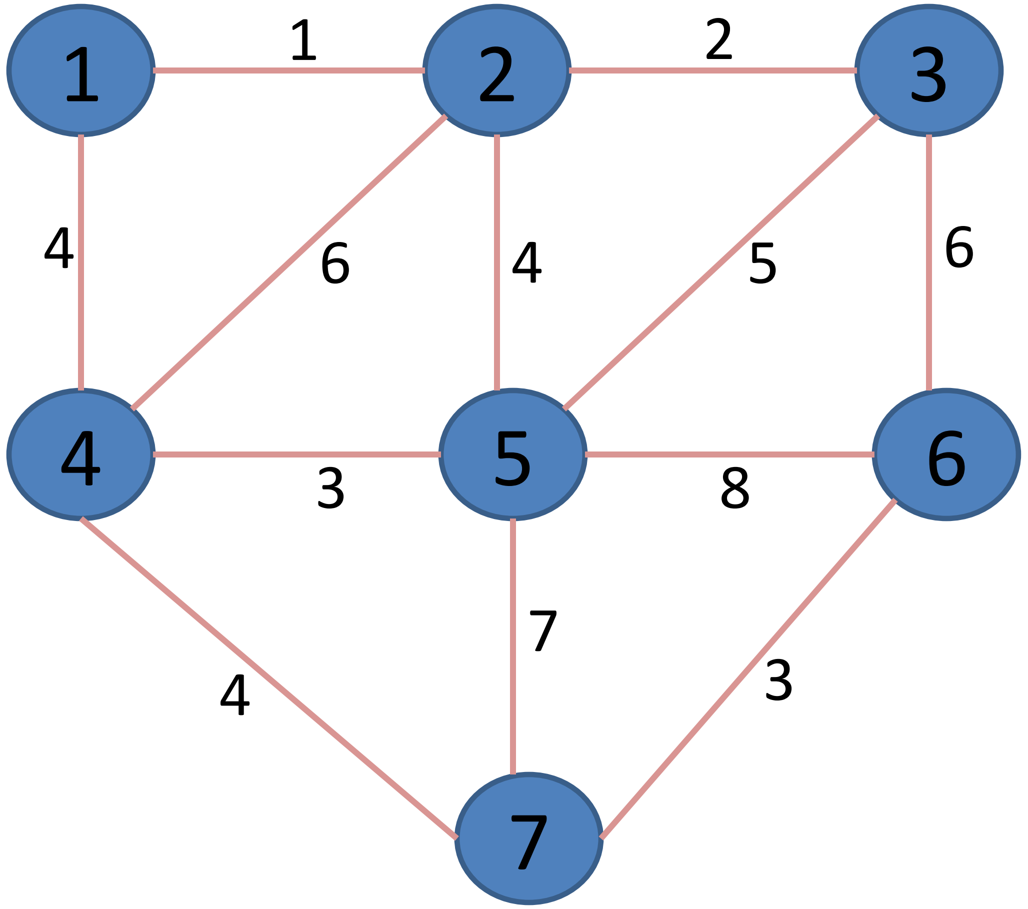
- Find the minimum spanning tree for the following graph using Prim's Algorithm.



Prim’s Algorithm - Example

Step:2

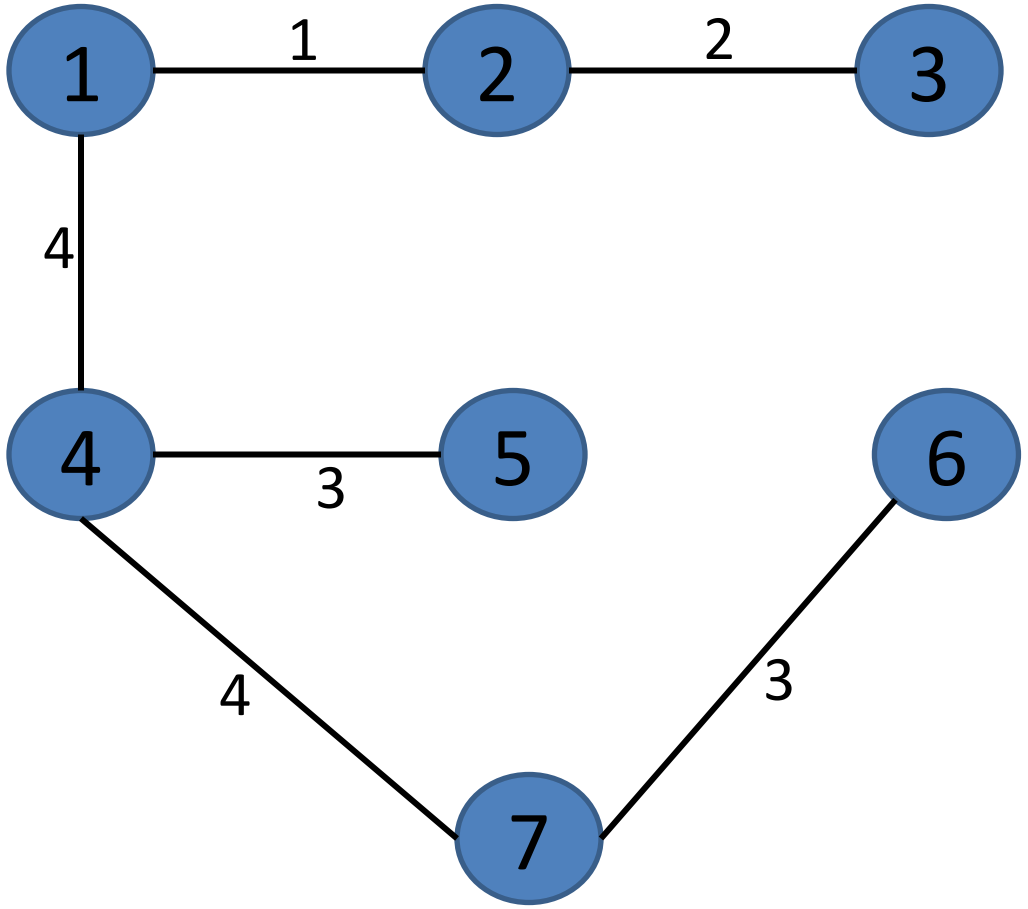
Find an edge with minimum weight



Node	Edges

Prim's Algorithm - Example

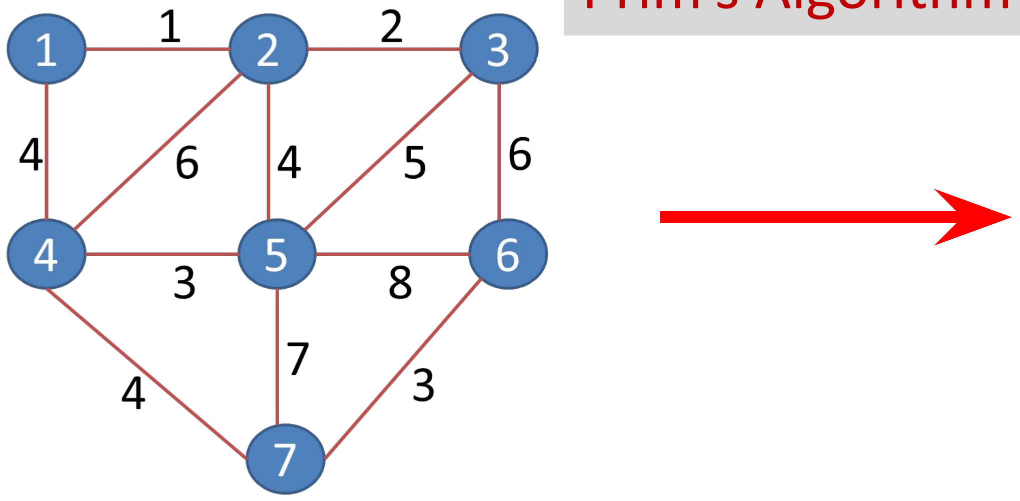
The minimum spanning tree for the given graph

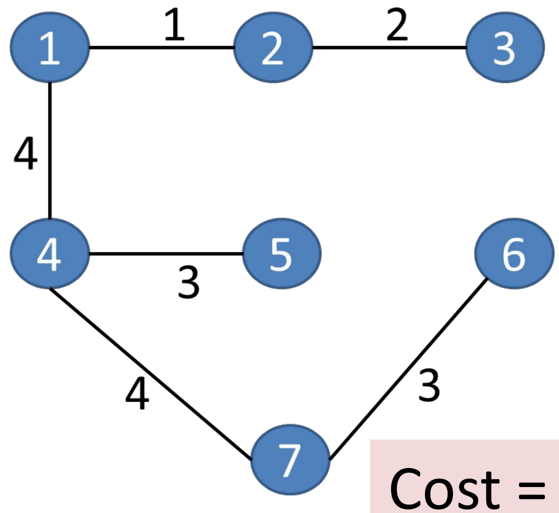


Node	Edges
1	{1, 2}
1, 2	{2, 3}
1, 2, 3	{1, 4}
1, 2, 3, 4	{4, 5}
1, 2, 3, 4, 5	{4, 7}
1, 2, 3, 4, 5, 6	{6, 7}

Total Cost = 17

Prim's Algorithm





Cost = 17

Step	Edge Selected {u, v}	Set B	Edges Considered
Init.	-	{1}	--
1	{1, 2}	{1,2}	**{1,2}** {1,4}
2	{2, 3}	{1,2,3}	{1,4} **{2,3}** {2,4} {2,5}
3	{1, 4}	{1,2,3,4}	**{1,4}** {2,4} {2,5} {3,5} {3,6}
4	{4, 5}	{1,2,3,4,5}	{2,4} {2,5} {3,5} {3,6} **{4,5}** {4,7}
5	{4, 7}	{1,2,3,4,5,7}	{2,4} {2,5} {3,5} {3,6} **{4,7}** {5,6} {5,7}
6	{6,7}	{1,2,3,4,5,6,7}	{2,4} {2,5} {3,5} {3,6} {5,6} {5,7} **{6,7}**

Prim’s Algorithm

```
Function Prim(G = (N, A): graph; length: A — R+):
set of edges
T ← ∅
B ← {an arbitrary member of N}
while B ≠ N do
     find e = {u, v} of minimum length such that
          u ∈ B and v ∈ N \ B
     T ← T U {e}
     B ← B U {v}
return T
```

Single Source Shortest Path – Dijkstra’s Algorithm

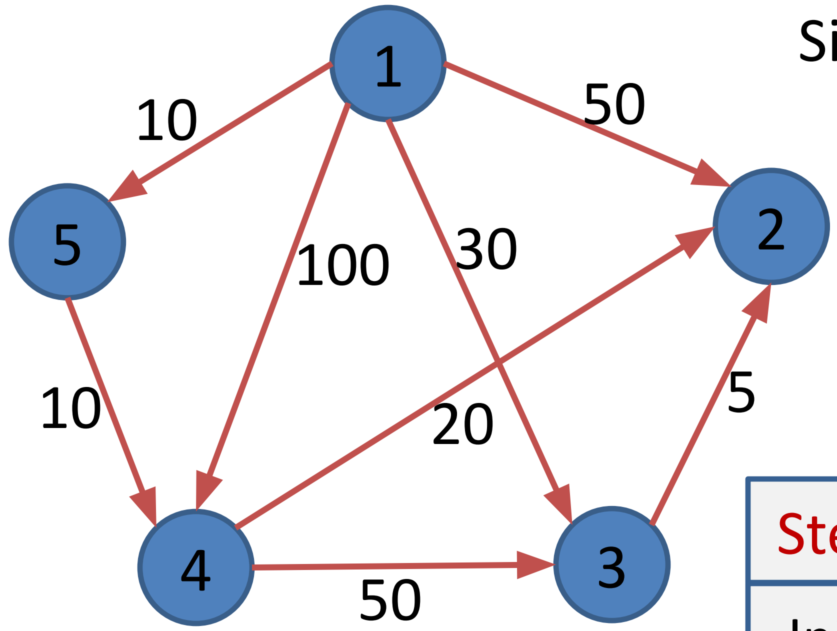
Dijkstra’s Algorithm

- Consider now **a directed graph $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{A})$** where N is the set of nodes and A is the set of directed edges of graph G.
- Each edge has a **positive length**.
- One of the nodes is designated as the **source node**.
- The problem is **to determine the length of the shortest path** from the source to each of the other nodes of the graph.
- The **algorithm maintains a matrix $\boldsymbol{L}$** which gives the length of each directed edge:

$$L[i,j] \geq 0 \textit{ if the edge } (i,j) \in A, \textit{and}$$
$$L[i,j] = \infty \textit{ otherwise.}$$

Dijkstra’s Algorithm - Example

Single source shortest path algorithm



Source node = 1

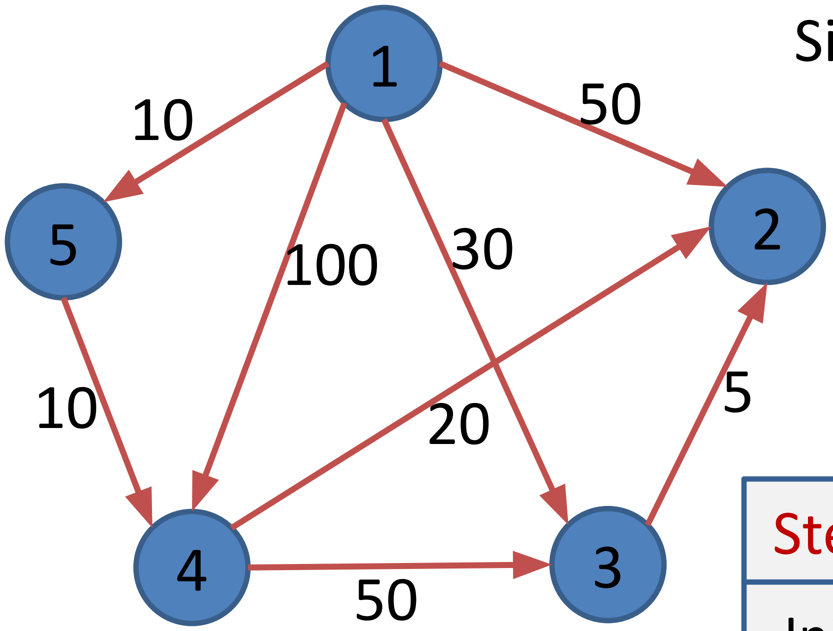
Step	v	C	2	3	4	5
Init.	-	{2, 3, 4, 5}	50	30	100	$\boxed{10}$
1	5	{2, 3, 4}	50	30	20	10

Is there path from 1 - 5 - 4

No Yes

Compare cost of 1 – 5 – 4 and 1- 4

Dijkstra's Algorithm - Example



Single source shortest path algorithm

Source node = 1

Step	v	C	2	3	4	5
Init.	-	{2, 3, 4, 5}	50	30	100	10
1	5	{2, 3, 4}	50	30	$\boxed{20}$	10
2	4	{2, 3}	40	$\boxed{30}$	20	10
3	3	{2}	35	30	20	10

Is there path from 1 - 4 - 5

No Yes

Compare cost of 1 – 4 – 3 and 1-3

Dijkstra’s Algorithm

```
Function Dijkstra(L[1 .. n, 1 .. n]): array [2..n]
array D[2.. n]
C ← {2,3,…, n}
{S = N \ C exists only implicitly}
for i ← 2 to n do
  D[i] ← L[1, i]
repeat n - 2 times
  v ← some element of C minimizing D[v]
  C ← C \ {v} {and implicitly S ← S U {v}}
  for each w ∈ C do
    D[w] ← min(D[w], D[v] + L[v, w])
return D
```

Comparison

- Dijkstra’s algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node

- Kruskal’s algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge

- Prim’s algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree

Fractional Knapsack Problem

Fractional Knapsack Problem

- Fractional knapsack allows taking fractions of items, maximizing value while staying within a knapsack's weight limit.
- Greedy algorithm sorts items by value-to-weight ratio and selects in descending order.
- Optimal solution with O(n log n) time complexity.

Fractional Knapsack Problem

- **Advantages:-**

1. **Optimal Solution:** The greedy algorithm used to solve the fractional knapsack problem provides an optimal solution, ensuring the highest possible total value of items taken within the knapsack's weight capacity.
2. **Continuous Solutions:** The fractional nature of the problem allows for continuous solutions, making it applicable in scenarios where items can be divided or used partially, like in some resource allocation or cutting stock problems.
3. **Real-world Applications:** Fractional knapsack has practical applications in various industries, such as resource optimization, finance, and production planning, where resources need to be allocated efficiently to maximize returns.

Fractional Knapsack Problem

- **Applications:-**

1. **Resource Management:** Fractional knapsack is used in resource management and allocation problems, where resources like manpower, budget, or time need to be allocated efficiently among different projects or tasks to maximize overall productivity or profit.

2. **Portfolio Optimization:** In finance, fractional knapsack algorithms can be applied to portfolio optimization, where investors need to decide how much of each asset to include in their investment portfolio to achieve the highest possible return while managing risk.

Fractional Knapsack Problem

- **Applications:- (Cont.)**

3. Cloud Resource Allocation: Cloud service providers use fractional knapsack algorithms to allocate resources like CPU, memory, and storage efficiently among different users or applications to ensure maximum utilization of cloud resources.

4. Data Compression: Fractional knapsack techniques are used in data compression algorithms, where the goal is to represent data using the least amount of storage space while preserving its essential characteristics.

5. Wireless Communication: In wireless communication systems, fractional knapsack algorithms can be utilized to optimize data transmission rates, ensuring the most critical data is transmitted with higher priority.

Fractional Knapsack Problem (Cont.)

- **Applications:- (Cont.)**

6. Time Management: In personal time management, fractional knapsack principles can be applied to prioritize tasks and activities to make the most of available time and achieve maximum productivity

Fractional Knapsack Problem

- We are given n objects and a knapsack.
- Object $\boldsymbol{i}$ has a positive weight $\boldsymbol{w_i}$ and a positive value $\boldsymbol{v_i}$ for $\boldsymbol{i} = \mathbf{1}, \mathbf{2} \ldots \boldsymbol{n}$.
- The knapsack can carry a weight not exceeding $\boldsymbol{W}$.
- Our aim is to fill the knapsack in a way that **maximizes** the value of the included objects, while respecting the capacity constraint.

Fractional Knapsack Problem

- In a fractional knapsack problem, we assume that the objects **can be broken into smaller pieces**.
- So we may decide to carry only a fraction $\boldsymbol{x_i}$ of object $\boldsymbol{i}$, where $\mathbf{0 \leq x_i \leq 1}$.
- In this case, object $\boldsymbol{i}$ contribute $\boldsymbol{x_i w_i}$ to the total weight in the knapsack, and $\boldsymbol{x_i v_i}$ to the value of the load.
- Symbolic Representation of the problem can be given as follows:

maximize $\boxed{\sum_{i=1}^{n} x_i v_i}$ **subject to** $\underline{\sum_{i=1}^{n} x_i w_i \leq W}$

Where, $v_i > 0$, $w_i > 0$ and $0 \leq x_i \leq 1$ for $1 \leq i \leq n$.

Fractional Knapsack Problem - Example

- Example: We are given 5 objects and the weight carrying capacity of knapsack is $W = 100.$
- For each object, weight w_i and value v_i are given in the following table.

- Fill the knapsack with given objects such that the total value of knapsack is **maximized.**

Greedy Solution

- Three **selection functions** can be defined as,
  1. Sort the items in **descending order of their values** and select the items till weight criteria is satisfied.
  2. Sort the items in **ascending order of their weight** and select the items till weight criteria is satisfied.
  3. To calculate the **ratio value/weight** for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add it.

Fractional Knapsack Problem - Solution

Selection	Objects					Value
	1	2	3	4	5	

Weight Capacity 100

30	50	20	
10	20	30	40
30	10	20	40

Profit = 66 + 20 + 30 + 48 = 164

Fractional Knapsack Problem - Algorithm

```
Algorithm: Greedy-Fractional-Knapsack (w[1..n],
p[1..n], W)
for i = 1 to n do
      x[i] ← 0
      weight ← 0
While weight < W do
      i ← the best remaining object
      if weight + w[i] ≤ W then
            x[i] ←1
            weight ← weight + w[i]
      else
            x[i] ← (W - weight) / w[i]
            weight ← W
return x
```

(100 – 60) / 50
= 0.8

Activity Selection Problem

Activity Selection Problem

- An activity-selection is the problem of **scheduling a resource** among several competing activities.
- We are given a set S of n activities with start time $\boldsymbol{s_i}$ and finish time $\boldsymbol{f_i}$, of an $\boldsymbol{i^{th}}$ activity. **Find the maximum size set of mutually compatible activities.**
- Activities $\boldsymbol{i}$ and $\boldsymbol{j}$ are compatible if the half-open internal $[\boldsymbol{s_i}, \boldsymbol{f_i})$ and $[\boldsymbol{s_j}, \boldsymbol{f_j})$ do not overlap, that is, i and j are compatible if $\boldsymbol{s_i} \geq \boldsymbol{f_j}$ or $\boldsymbol{s_j} \geq \boldsymbol{f_i}$.

Activity Selection Problem - Example

- Example: 11 activities are given as,
- Solution:

1. Sort the activities of set ***S*** as per increasing finish time to directly identify mutually compatible activities by comparing **finish time of first activity and start time of next activity.**

Sr.	Activity	
1	P	(1, 4)
2	Q	(3, 5)
3	R	(0, 6)
4	S	(5, 7)
5	T	(3, 8)
6	U	(5, 9)
7	V	(6, 10)
8	W	(8, 11)
9	X	(8, 12)
10	Y	(2, 13)
11	Z	(12, 14)

Activity Selection Problem - Example

- Solution:

1. A = {P}
2. A = {P, S}
3. A = {P, S, W}
4. A = {P, S, W, Z}

Answer: A = {P, S, W, Z}

Sr.	Activity	
1	P	$\boxed{(1, 4)}$
~~2~~	~~Q~~	~~(3, 5)~~
3	R	(0, 6)
4	S	$\boxed{(5, 7)}$
5	T	(3, 8)
6	U	(5, 9)
7	V	(6, 10)
8	W	$\boxed{(8, 11)}$
9	X	(8, 12)
10	Y	(2, 13)
11	Z	$\boxed{(12, 14)}$

Activity Selection - Algorithm

```
Algorithm: Activity Selection
Step I: Sort the input activities by increasing
finishing time.  $f_1 \le f_2 \le \ldots \le f_n$

Step II: Call GREEDY-ACTIVITY-SELECTOR (s, f)
   n = length [s]
   A = {i}
   j = 1
   for  i = 2  to  n
         do if   $s_i \ge f_j$
              then  A = A U {i}
                        j = i
   return  set A
```

Job Scheduling with Deadlines

Job Scheduling with Deadlines

- We have set of $\boldsymbol{n}$ jobs to execute, each of which **takes unit time**.
- At any point of time we can **execute only one job.**
- Job $\boldsymbol{i}$ earns profit $g_i > \mathbf{0}$ if and only if it is executed **no later than** time d_i.
- We have to find an optimal sequence of jobs such that our total **profit is maximized**.
- *Feasible jobs: A set of job is feasible if there exits **at least one sequence** that allows all the jobs in the set to be executed no later than their respective deadlines.*

Job Scheduling with Deadlines - Example

- Using greedy algorithm find an optimal schedule for following jobs with $\boldsymbol{n = 6}$.
- Profits: $(P_1, P_2, P_3, P_4, P_5, P_6) = (\underleftarrow{\overrightarrow{15,20}},10,7,5,3)$ &
- Deadline: $(d_1, d_2, d_3, d_4, d_5, d_6) = (1, 3, 1, 3, 1, 3)$

Solution:

$\boxed{\text{Step 1:}}$ Sort the jobs in **decreasing order** of their profit.

Job Scheduling with Deadlines - Example

Step 2: Find total position $P = \min(n, \max(d_i))$

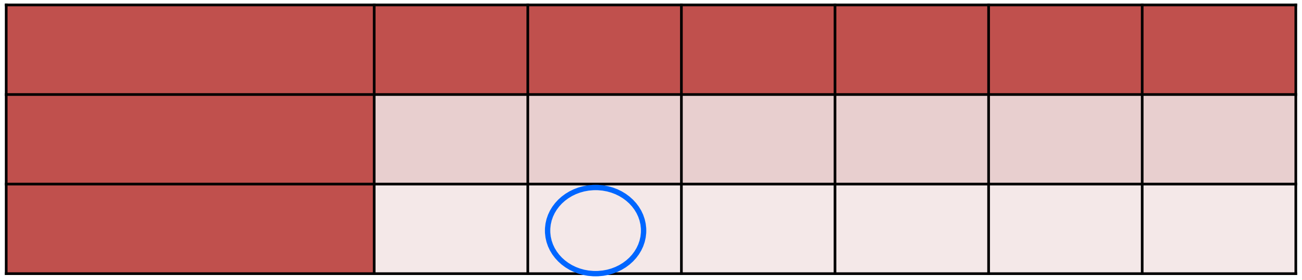
Here, $P = \min(6, 3) = \mathbf{3}$

P	1	2	3
Job selected	0	0	0

Step 3: $d_1 = 3$: assign job 1 to position 3

P	1	2	3
Job selected	0	0	J1

Job Scheduling with Deadlines - Example



Step 4: $d_2 = 1$: assign job 2 to position 1

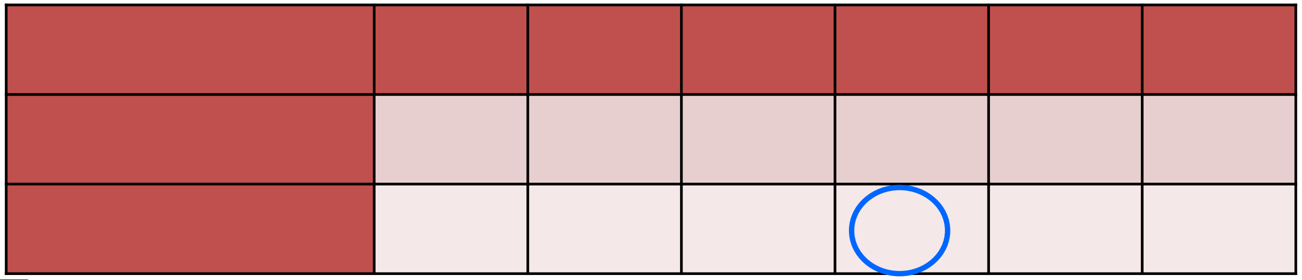
P	1	2	3
Job selected	J2	0	J1

Step 5: $d_3 = 1$: assign job 3 to position 1

Position 1 is already occupied, so reject job 3

Job Scheduling with Deadlines - Example

▪



$\boxed{\text{Step 6:}}$ $d_4 = 3$: assign job 4 to position 2 as, position 3 is not free but position 2 is free.

P	1	2	3
Job selected	J2	J4	J1

- Now **no more free position** is left so no more jobs can be scheduled.
- The final optimal sequence:

Execute the job in order 2, 4, 1 with total profit value 42.

Job Scheduling with Deadlines

- Time complexity:
  - Worst case :- O(n^2)
  - Best Case: O(n^2)
  - Average case: O(n)
- It has uniprocessor system, which means Only one processor is available for processing all the jobs.
- The profit of a job is given only when that job is completed within its deadline
- Job should be completed, we cannot break the job in between the task.
- Processor takes one unit of time to complete a job. (unit eg. Month, day)

Job Scheduling with Deadlines- Example 2

- **Given the jobs, their deadlines and associated profits as shown-**

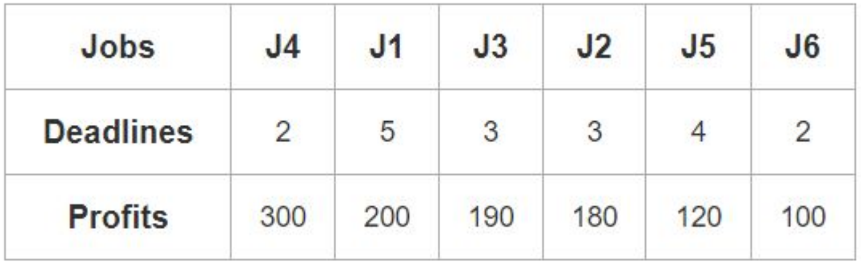
1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Job Scheduling with Deadlines-Example 2

Step-01:

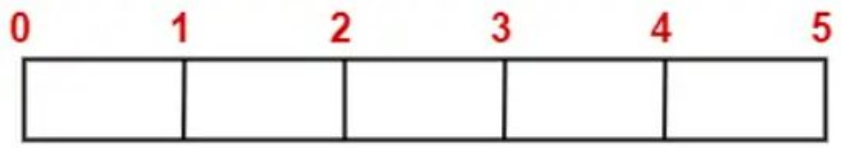
Sort all the given jobs in decreasing order of their profit-



Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

- Value of maximum deadline = 5.
- So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-

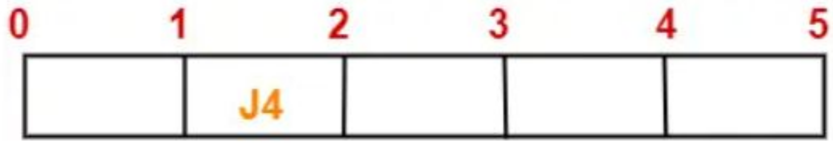


Gantt Chart

Job Scheduling with Deadlines-Example 2

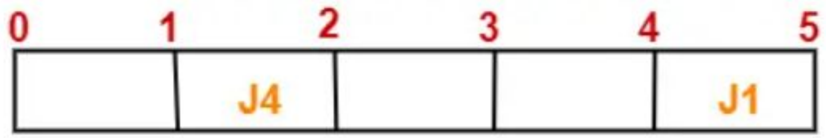
Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



Step-04:

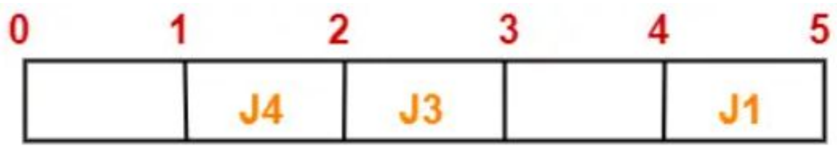
- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Job Scheduling with Deadlines- Example 2

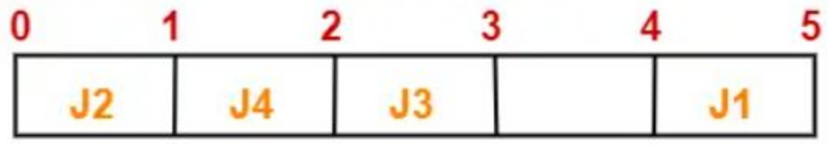
Step-05:

- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



Step-06:

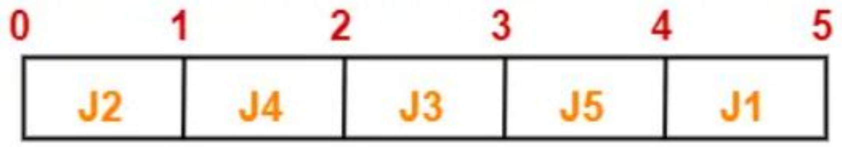
- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-



Job Scheduling with Deadlines- Example 2

Step-07:

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



- Now,
- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

Job Scheduling with Deadlines- Example 2

- Now, the given questions may be answered as-

1. **Write the optimal schedule that gives maximum profit.**
   - The optimal schedule is- **J2 , J4 , J3 , J5 , J1**
2. **Are all the jobs completed in the optimal schedule?**
   - All the jobs are not completed in optimal schedule.
   - This is because job J6 could not be completed within its deadline.
3. **What is the maximum earned profit?**

   Maximum earned profit

   = Sum of profit of all the jobs in optimal schedule

   = Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

   = 180 + 300 + 190 + 120 + 200

   = 990 units

https://www.gatevidyalay.com/job-sequencing-with-deadlines/#google_vignette

Job Scheduling with Deadlines - Algorithm

```
Algorithm: Job-Scheduling (P[1..n], D[1..n])
1.  Sort all the n jobs in decreasing order of their
    profit.
2.  Let total position P = min(n, max($d_i$))
3.  Each position 0, 1, 2..., P is in different set and
    T({i}) = i, for 0 ≤ i ≤ P.
4.  Find the set that contains d, let this set be K. if
    T(K) = 0 reject the job; otherwise:
    a.  Assign the new job to position T(K).
    b.  Find the set that contains T(K) – 1. Call this set L.
    c.  Merge K and L. the value for this new set is the old
        value of T(L).
```

Huffman Codes

Huffman Codes

- Huffman invented a greedy algorithm that constructs **an optimal prefix code** called a Huffman code.
- Huffman coding is a **lossless data compression** algorithm.
- It assigns **variable-length codes** to input characters.
- Lengths of the assigned codes are based on the **frequencies of corresponding characters**.
- The most frequent character gets the smallest code and the least frequent character gets the largest code.
- The variable-length codes assigned to input characters are Prefix Codes.
- Overall time complexity of Huffman Coding is **O(nlogn)**

Huffman Codes

- In Prefix codes, the codes are assigned in such a way that the code assigned to one character **is not a prefix of code** assigned to any other character.
- For example,
- a = $\underline{01}$, b = $\boxed{01}0$ and c = 11 Not a prefix code
- This is how Huffman Coding makes sure that there is **no ambiguity** when decoding the generated bit stream.
- There are mainly two major parts in Huffman Coding
  1. Build a Huffman Tree from input characters.
  2. Traverse the Huffman Tree and assign codes to characters.

Huffman Codes-Rules

- If you assign weight ‘0’ to the left edges, then assign weight ‘1’ to the right edges.
- If you assign weight ‘1’ to the left edges, then assign weight ‘0’ to the right edges.
- Any of the above two conventions may be followed.
- But follow the same convention at the time of decoding that is adopted at the time of encoding.

Huffman Codes - Example

- Find the Huffman codes for the following characters.

Characters	a	b	c	d	e	f
Frequency (in thousand)	45	13	12	16	9	5

Step 1:

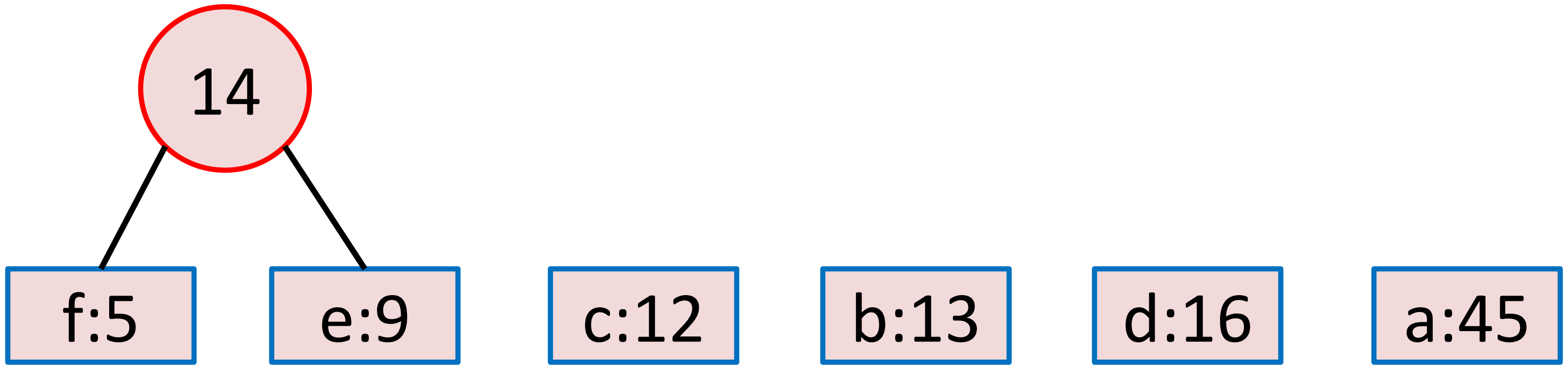
Arrange the characters in the Ascending order of their frequency.

f:5	e:9	c:12	b:13	d:16	a:45

Huffman Codes - Example

Step 2:

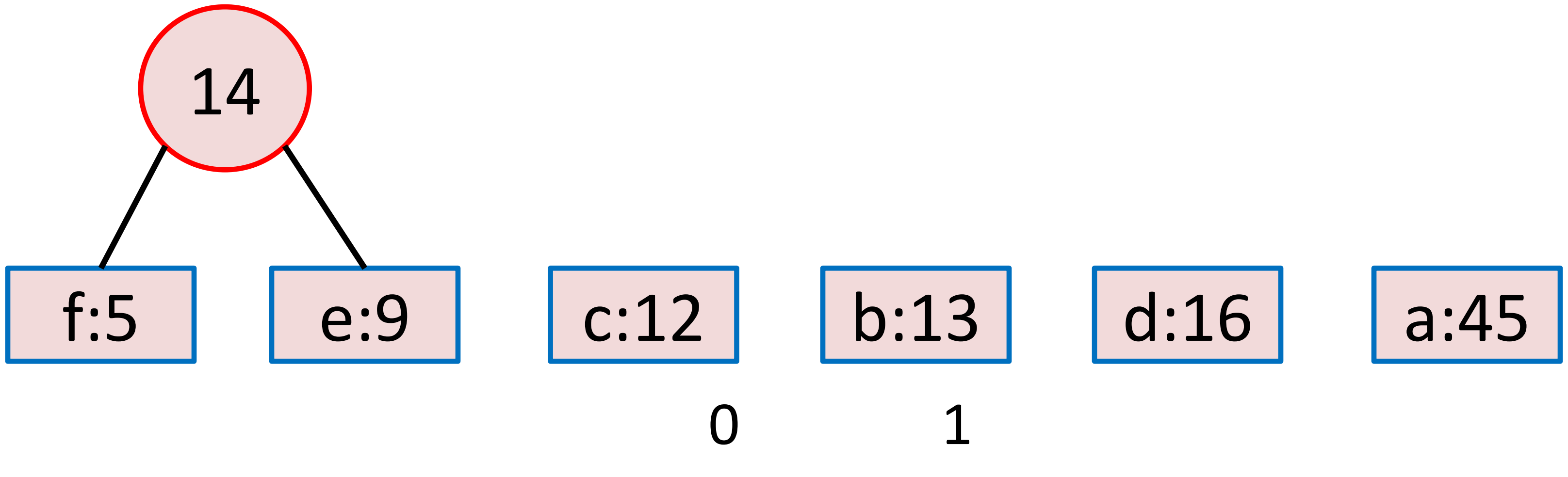
- Extract two nodes with the minimum frequency.
- Create a new internal node with frequency equal to the sum of the two nodes frequencies.
- Make the first extracted node as its left child and the other extracted node as its right child.



Huffman Codes - Example

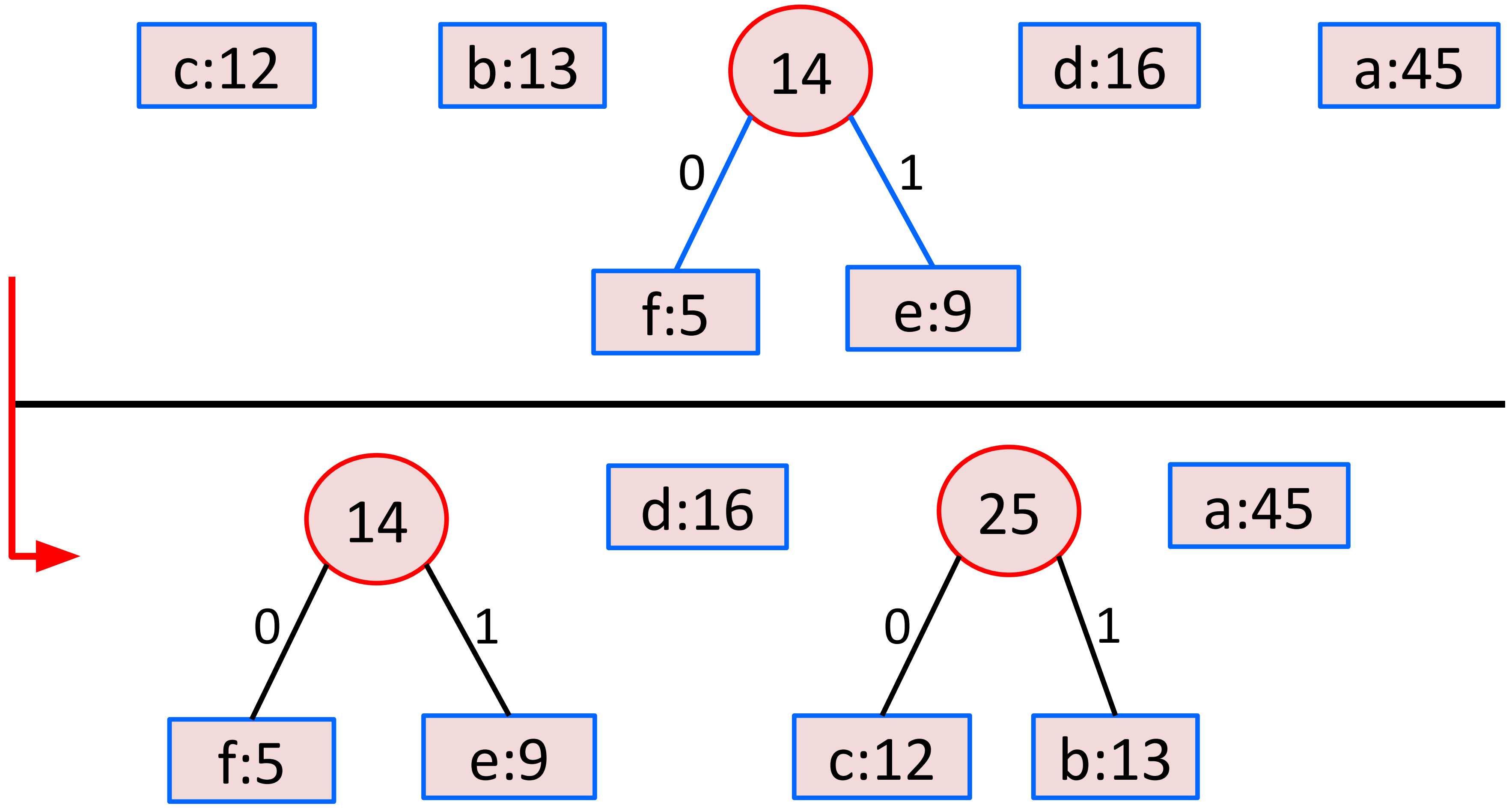
Step 3:

- Rearrange the tree in ascending order.
- Assign 0 to the left branch and 1 to the right branch.
- Repeat the process to complete the tree.



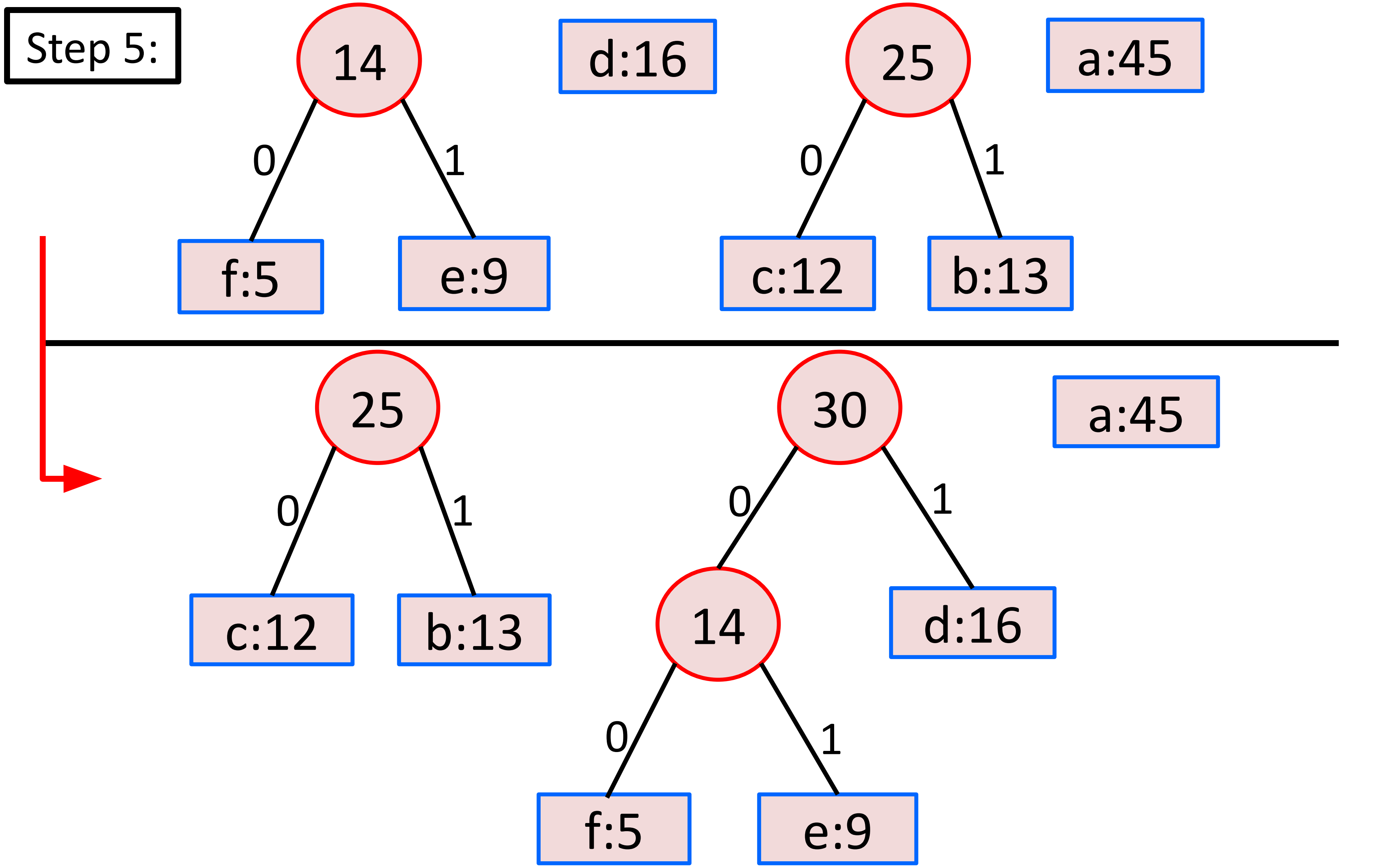
Huffman Codes - Example

Step 4:



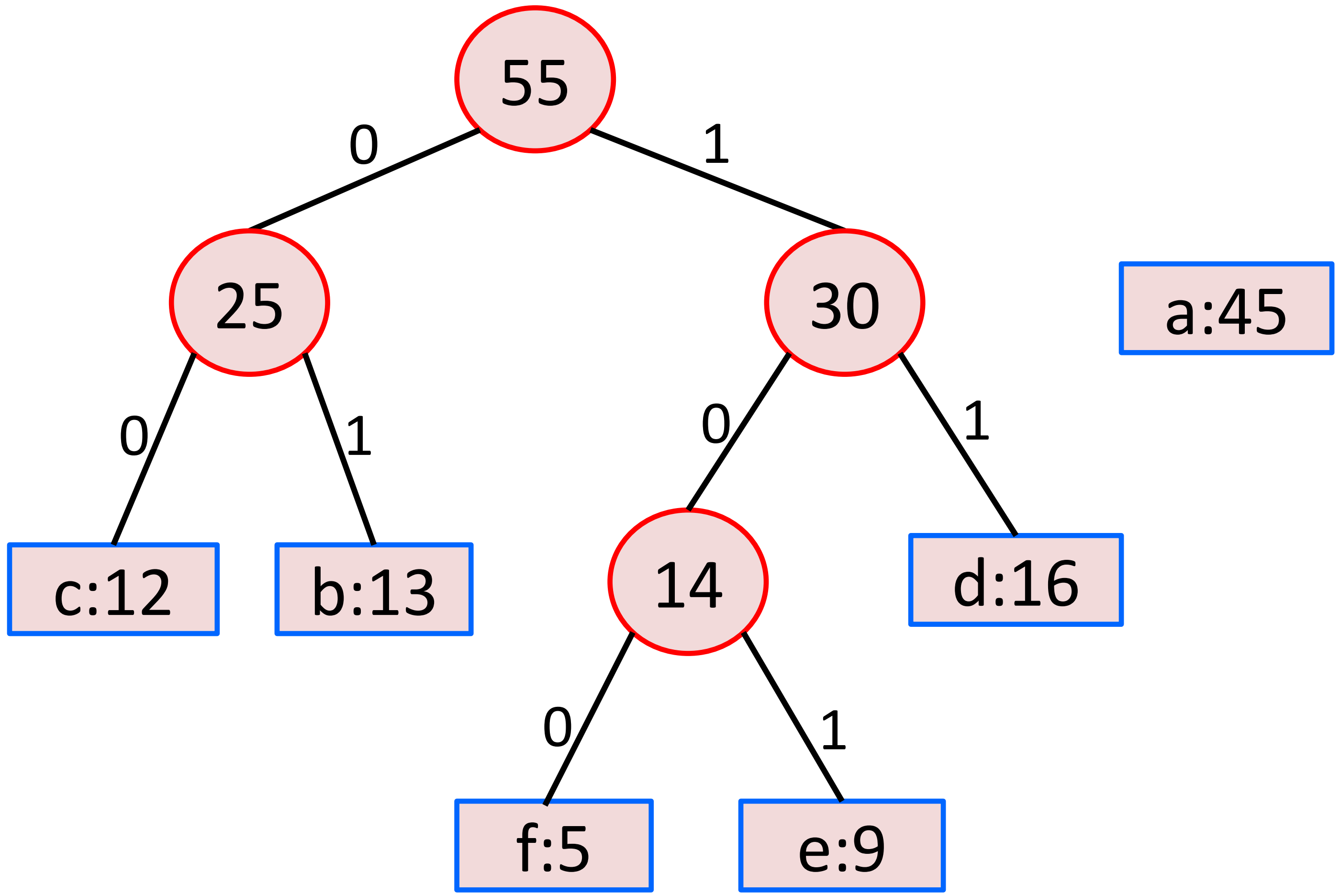
Huffman Codes - Example

Step 5:



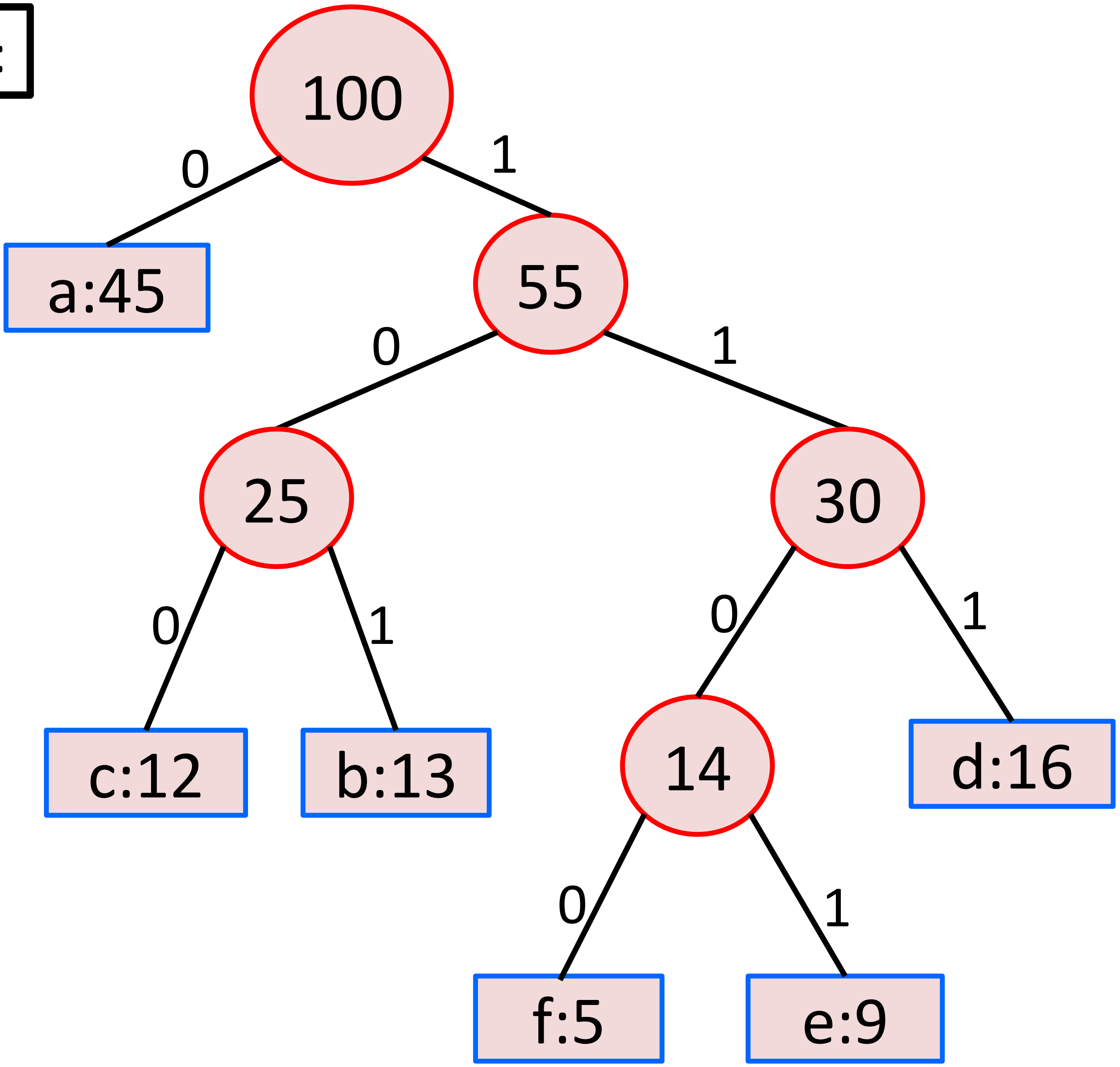
Huffman Codes - Example

Step 6:



Huffman Codes - Example

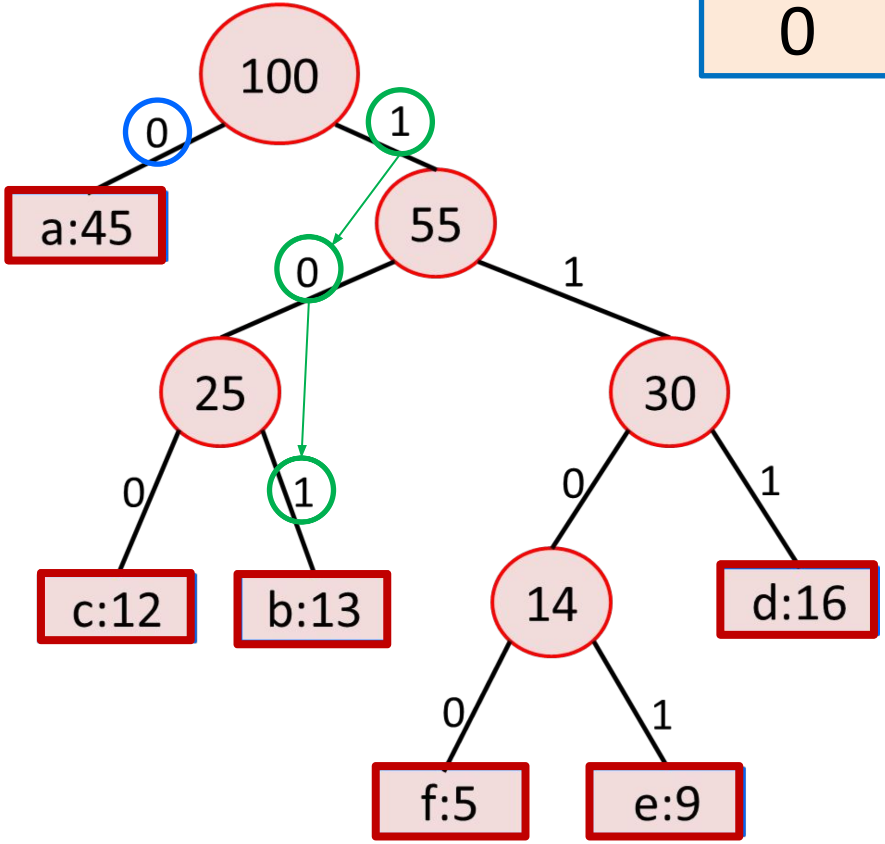
Step 7:



Huffman Codes - Example

Step 8:

Characters	a	b	c	d	e	f
Frequency (in thousand)	45	13	12	16	9	5
	0	101	100	111	1101	1100



Huffman Codes - Example

1. **Huffman Code For Characters**

Characters	a	b	c	d	e	f
Frequency (in thousand)	45	13	12	16	9	5
	0	101	100	111	1101	1100

2. **Average Code Length**

Average code length= $\sum$ (frequency$_i$ x code length$_i$) / $\sum$ (frequency$_i$)

={(45x01)+(13x3)+(12x3)+(16x3)+(9x4)+(5x4)}/(45+13+12+16+9+5)

= {45+39+36+48+36+20}/100

=224/100

=2.24

Huffman Codes - Example

3. <u>Length of Huffman Encoded Message</u>

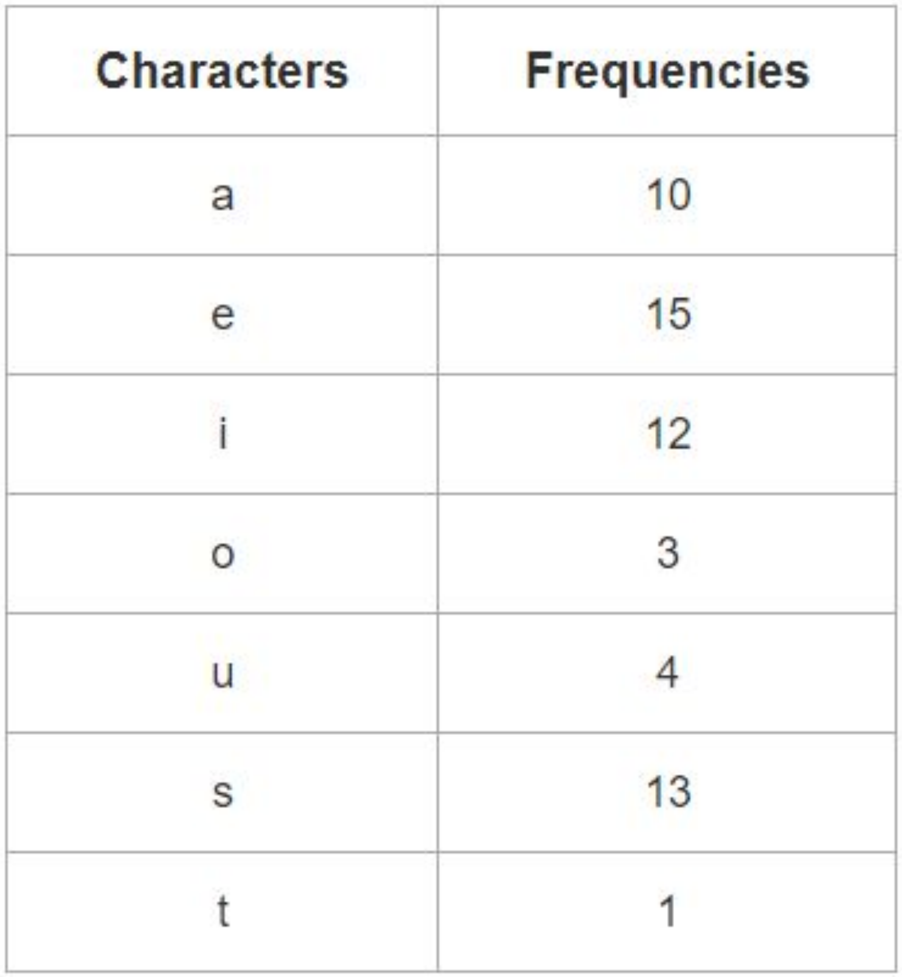
Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

=100*2.24

=224

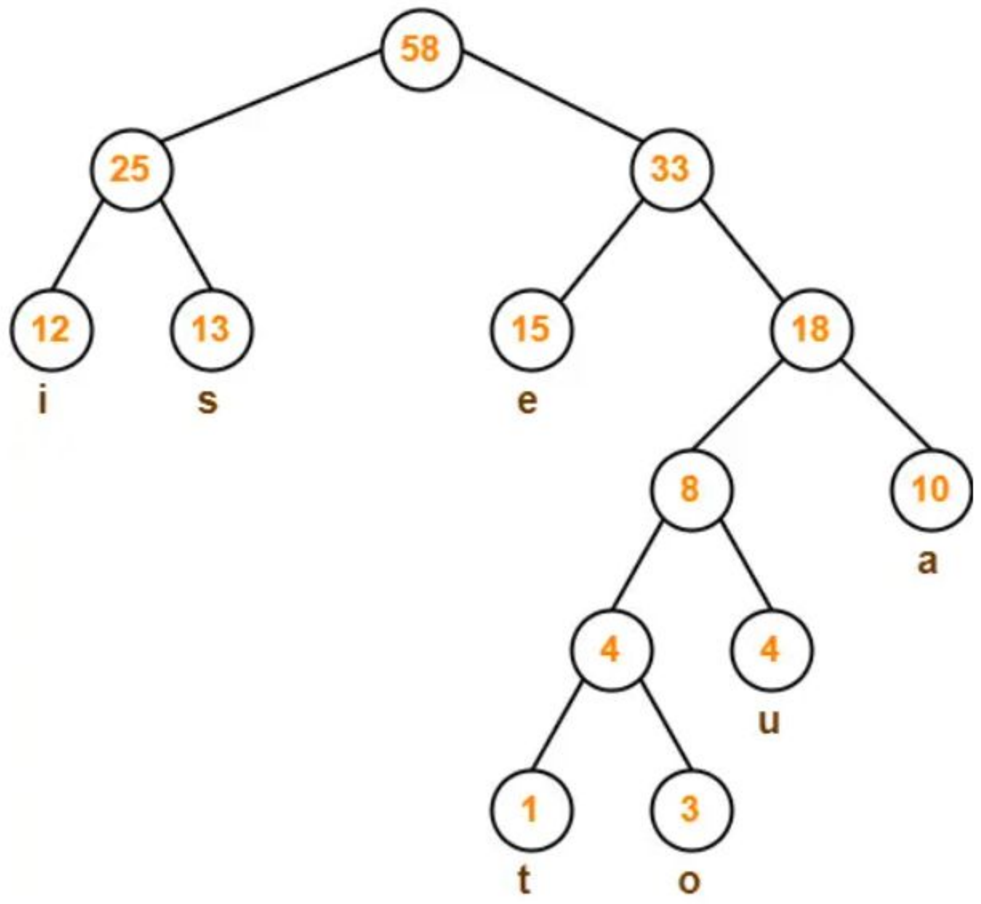
Huffman Codes – Example -2



Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

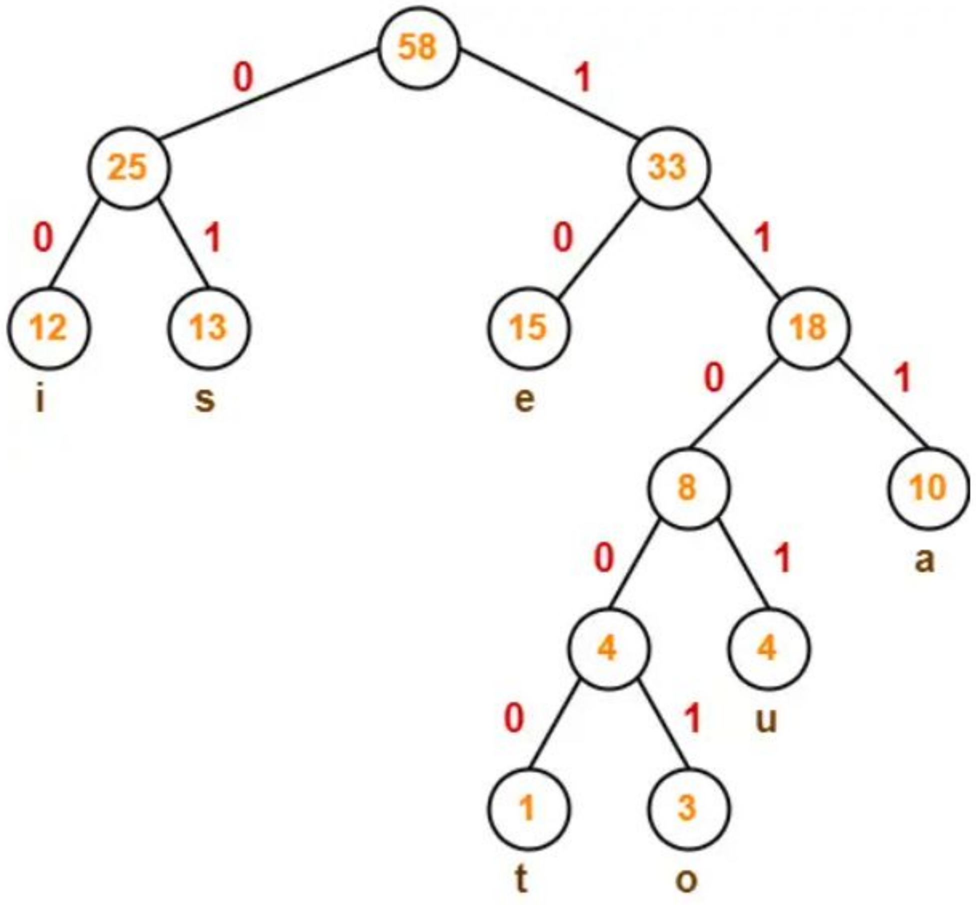
Huffman Codes – Example -2

Final graph



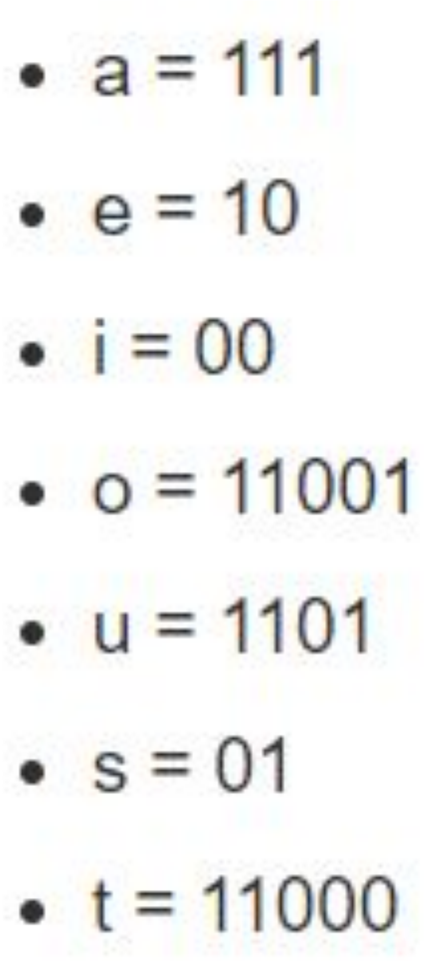
Huffman Codes – Example -2

- Final Graph with weight



Huffman Codes – Example -2

- Final Graph.



- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

Huffman Codes - Algorithm

```
Algorithm: HUFFMAN (C)
n = |C|
Q = C
for i = 1 to n-1
        allocate a new node z
        z.left = x = EXTRACT-MIN(Q)
        z.right = y = EXTRACT-MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q,z)
return EXTRACT-MIN(Q)  // return the root of the tree
```